Table of Contents

Part 3 of 3 - Explore all 232+ Linux commands at dargslan.com/learn/linux-commands
Each command includes syntax, options, practical examples with output, and pro tips.

$ tail

Beginner

Output the last part of files

The tail command outputs the last part of files. By default, it prints the last 10 lines. Its most important feature is the -f (follow) mode, which continuously monitors a file for new content - making it indispensable for real-time log monitoring.

tail is a critical tool for system administrato...

Options & Flags

`-n`	Output the last N lines (default 10)
`-f`	Follow - output appended data as the file grows
`-F`	Follow with retry (handles log rotation)
`-c`	Output the last N bytes
`-n +N`	Output starting from line N
`-q`	Quiet mode - suppress headers when following multiple files
`--pid`	Terminate tail after process with PID exits

Practical Examples

Example: View last 10 lines

```
$ tail /var/log/auth.log
```

Shows the last 10 lines of the authentication log.

Example: Monitor a log file in real-time

```
$ tail -f /var/log/nginx/access.log
```

Continuously shows new lines as they are written. Press Ctrl+C to stop.

Example: Follow with log rotation support

```
$ tail -F /var/log/syslog
```

Like -f but re-opens the file if it is rotated or recreated. Essential for production servers.

Example: Skip CSV header

```
$ tail -n +2 data.csv
```

Outputs everything starting from line 2, effectively removing the header row.

Example: View last N lines

```
$ tail -n 100 error.log
```

Shows the last 100 lines of the error log.

Tips & Best Practices

Pro Tip: Use -F in production: Always use -F instead of -f in production environments where log rotation (logrotate) is active. -F re-opens the file if it is rotated.

Note: multitail for advanced monitoring: For monitoring multiple logs with colors and filtering, consider multitail - it provides a split-screen view of multiple log files.

Warning: Buffer flushing with pipes: When piping tail -f through grep, add --line-buffered to grep to prevent output buffering delays.

$ tee

Intermediate

Read from stdin and write to stdout and files

The tee command reads from standard input and writes to both standard output and one or more files simultaneously. Named after the T-splitter used in plumbing, it splits the output stream so you can save a copy while also passing data to the next command in a pipeline.

tee is essential when you ...

Options & Flags

`-a`	Append to files instead of overwriting
`-i`	Ignore interrupt signals (SIGINT)
`--output-error`	Set behavior on write error (warn, exit, etc.)
`multiple files`	Write to multiple files simultaneously
`/dev/null`	Discard stdout while keeping file output
`>(cmd)`	Process substitution - send to another command

Practical Examples

Example: Save and display output

```
$ ls -la | tee directory_listing.txt
```

Shows the directory listing on screen AND saves it to a file.

Example: Append to log file

```
$ echo "Deploy complete at $(date)" | tee -a deploy.log
```

Appends a timestamped message to the log file while displaying it.

Example: Write to protected file with sudo

```
$ echo '127.0.0.1 mysite.local' | sudo tee -a /etc/hosts
```

The standard way to append to root-owned files. sudo echo > file does NOT work.

Example: Save intermediate pipeline result

```
$ cat access.log | tee raw.log | grep 'ERROR' | tee errors.log | wc -l
42
```

Captures raw and filtered data at different stages of the pipeline.

Example: Write to multiple files

```
$ date | tee file1.txt file2.txt file3.txt
```

Writes the same output to three files simultaneously.

Tips & Best Practices

Pro Tip: sudo tee for root files: Use echo "content" | sudo tee /etc/file instead of sudo echo "content" > /etc/file. The redirect runs as your user, not root.

Note: Capture stderr too: To capture both stdout and stderr: command 2>&1 | tee output.log. This redirects stderr to stdout before tee.

Warning: Overwrite by default: tee overwrites files by default. Always use -a (append) for log files to avoid losing previous content.

$ tr

Intermediate

Translate or delete characters

The tr (translate) command replaces, deletes, or squeezes characters in text from standard input. It operates on individual characters (not strings), making it perfect for simple character-level transformations.

tr reads from stdin only - it cannot read files directly, so it is always used with ...

Options & Flags

`-d`	Delete characters in SET1
`-s`	Squeeze repeated characters to single occurrence
`-c`	Complement SET1 (use all characters NOT in SET1)
`[:upper:]`	Character class: uppercase letters
`[:lower:]`	Character class: lowercase letters
`[:digit:]`	Character class: digits 0-9

Practical Examples

Example: Convert to uppercase

```
$ echo 'hello world' | tr '[:lower:]' '[:upper:]'
HELLO WORLD
```

Converts all lowercase letters to uppercase.

Example: Convert to lowercase

```
$ echo 'HELLO' | tr '[:upper:]' '[:lower:]'
hello
```

Converts all uppercase letters to lowercase.

Example: Replace spaces with newlines

```
$ echo 'one two three' | tr ' ' '\n'
one
two
three
```

Splits space-separated words into separate lines.

Example: Delete all digits

```
$ echo 'Order #12345' | tr -d '[:digit:]'
Order #
```

Removes all numeric characters from the input.

Example: Squeeze multiple spaces

```
$ echo 'too    many   spaces' | tr -s ' '
too many spaces
```

Collapses multiple consecutive spaces into a single space.

Tips & Best Practices

Warning: tr works on characters, not strings: tr 'abc' 'xyz' replaces a->x, b->y, c->z character by character. It does NOT replace the string 'abc' with 'xyz'. Use sed for string replacement.

Pro Tip: Generate random strings: tr -dc 'A-Za-z0-9' < /dev/urandom | head -c 32 generates a random 32-character alphanumeric string.

Note: Stdin only: tr cannot read files directly. Use: tr ... < file.txt or cat file.txt | tr ... to process files.

$ uniq

Beginner

Report or filter out repeated lines

The uniq command filters or reports adjacent repeated lines in sorted input. It can remove duplicates, count occurrences, show only duplicated lines, or show only unique lines.

Important: uniq only detects adjacent duplicates, so input must be sorted first (using sort) unless you specifically wa...

Options & Flags

`-c`	Prefix lines with occurrence count
`-d`	Only print duplicate lines
`-u`	Only print unique (non-duplicated) lines
`-i`	Case-insensitive comparison
`-f N`	Skip first N fields when comparing
`-s N`	Skip first N characters when comparing
`-w N`	Compare only first N characters

Practical Examples

Example: Remove duplicate lines

```
$ sort file.txt | uniq
```

Sorts and removes adjacent duplicates. This is the standard deduplication pattern.

Example: Count occurrences

```
$ sort access.log | uniq -c | sort -rn | head -10
    247 GET /api/status\n    189 GET /index.html
```

Finds the 10 most common lines in a log file with counts.

Example: Find duplicate entries

```
$ sort emails.txt | uniq -d
```

Shows only lines that appear more than once.

Example: Find unique-only entries

```
$ sort data.txt | uniq -u
```

Shows only lines that appear exactly once (no duplicates).

Example: Top IP addresses from access log

```
$ awk '{print $1}' access.log | sort | uniq -c | sort -rn | head -20
   1523 192.168.1.100\n    847 10.0.0.50
```

Extracts IPs, counts unique occurrences, and shows the top 20.

Tips & Best Practices

Warning: Always sort first: uniq only removes ADJACENT duplicates. Always pipe through sort first: sort file | uniq. Without sort, non-adjacent duplicates will remain.

Pro Tip: Frequency analysis pattern: The classic pattern: sort | uniq -c | sort -rn | head -N gives you the top N most frequent items in any list.

Note: sort -u as alternative: For simple deduplication, sort -u is more efficient than sort | uniq since it does both in a single pass.

$ wc

Beginner

Print newline, word, and byte counts for files

The wc (word count) command counts lines, words, characters, and bytes in files or input streams. Despite its name suggesting only word counting, it is most commonly used for counting lines.

wc is a fundamental tool in shell pipelines, often used to count results from grep, find, and other filte...

Options & Flags

`-l`	Count lines only
`-w`	Count words only
`-c`	Count bytes only
`-m`	Count characters (multi-byte aware)
`-L`	Print length of longest line
`--files0-from`	Read file list from a null-terminated input

Practical Examples

Example: Count lines in a file

```
$ wc -l /var/log/syslog
15243 /var/log/syslog
```

Shows the number of lines in the syslog file.

Example: Count files in directory

```
$ ls -1 /var/log | wc -l
47
```

Counts the number of files and directories in /var/log.

Example: Count matching lines

```
$ grep -c "ERROR" app.log
23
```

Count error occurrences (grep -c is equivalent to grep | wc -l but faster).

Example: Count words in a document

```
$ wc -w report.txt
2847 report.txt
```

Shows the total word count - useful for documents with word limits.

Example: Count lines of code

```
$ find src/ -name '*.php' | xargs wc -l | tail -1
12450 total
```

Counts total lines of PHP code in the src/ directory.

Tips & Best Practices

Pro Tip: Count without filename: To get just the number: wc -l < file.txt (redirect instead of argument) or cat file | wc -l.

Note: -c vs -m: -c counts bytes and -m counts characters. They differ for multi-byte encodings (UTF-8). Use -m for accurate character counts with non-ASCII text.

Warning: Trailing newline matters: wc -l counts newline characters. A file without a trailing newline will report one fewer line than you might expect.

$ xargs

Intermediate

Build and execute command lines from standard input

xargs builds and executes command lines from standard input. It reads items from stdin and passes them as arguments to a specified command, enabling you to use output from one command as arguments for another.

xargs solves the common problem of passing too many arguments to a command (argument l...

Options & Flags

`-I {}`	Replace {} with each input item
`-0`	Input items are null-terminated (use with find -print0)
`-n`	Use at most N arguments per command invocation
`-P`	Run up to N processes in parallel
`-p`	Prompt before execution (interactive)
`-t`	Print command before executing
`-L`	Use N lines per command invocation

Practical Examples

Example: Delete files found by find

```
$ find /tmp -name '*.tmp' -print0 | xargs -0 rm -f
```

Safely deletes files with special characters in names using null-terminated input.

Example: Count lines in all PHP files

```
$ find src/ -name '*.php' | xargs wc -l
```

Passes all PHP file paths as arguments to wc for line counting.

Example: Replace placeholder

```
$ find . -name '*.bak' | xargs -I {} mv {} {}.old
```

Renames each .bak file by appending .old using placeholder replacement.

Example: Parallel image processing

```
$ find photos/ -name '*.jpg' | xargs -P 8 -I {} convert {} -resize 800x600 resized/{}
```

Processes 8 images simultaneously for faster batch conversion.

Example: Batch chmod

```
$ find . -name '*.sh' | xargs chmod +x
```

Makes all .sh files executable in one efficient operation.

Tips & Best Practices

Warning: Always use -0 with find: Use find -print0 | xargs -0 to handle filenames with spaces, quotes, and newlines safely. Without -0, filenames with spaces break.

Pro Tip: Parallel execution: -P N runs N processes in parallel. Use -P $(nproc) to use all CPU cores. Great for batch image/video processing.

Note: xargs vs find -exec: xargs is faster than find -exec because it batches arguments. find -exec {} + also batches but xargs offers more features like parallel execution.